Model-Based Testing with Graph Transformation Systems

MSc Thesis *(Afstudeerscriptie)*

written by

Vincent de Bruijn

Formal Methods & Tools,
University of Twente,
Enschede,
The Netherlands

v.debruijn@student.utwente.nl

April 25, 2012

Abstract

Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process.This report describes the setup of a research project where the use of Graph Grammars in model-based testing is researched. The goal of the project is to create a system for automatic test generation from Graph Grammars and to validate this system. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system will be validated using the results of several case-studies.

Contents

Chapter 1

Introduction

In software development projects, often limited time and resources are available for testing. However, testing is an important part of software development, because it decreases future maintainance costs [14]. Testing is a complex process and should be done often [18]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of tests. However, when the creation of a test suite is done manually, this still leaves room for human error [11]. Also, manual creation of test-cases is not time-efficient.

Creating an abstract representation or a *model* of the system is a way to tackle these problems. What is meant by a model in this report, is the description of the behavior of a system. Models such as state charts and sequence charts, which only describe the system architecture, are not considered here. A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [13]. If the tester copies an error in the document or makes a wrong interpretation, the constructed model becomes incorrect.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency. The feedback process of the stakeholders is obstructed by low transparency models. Models that are often used are state machines, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The problem in such models with a large amount of states is the decrease of model transparency. Errors in models with a low transparency are not easily detected.

A formalism that among other things can describe software systems is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]: "Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules

are well-known examples." An informative paper on graph transformations is written by Heckel et al. [7]. A quote from this paper: "Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular."

The motivation is given for using graph grammars as a modelling technique. The goal of this research is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect models. The system design, once implemented and validated, provides a valuable contribution to the testing paradigm.

Tools that perform statespace exploration on graph grammars and tools for automatic test generation already exist. One tool for each of these functions will be used in this research. The graph transformation tool GROOVE[1] will be used to model and explore the graph grammar. The testing tool developed by Axini[2] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool will be referred to in this report as Axini Test Manager (ATM).

The structure of the rest of this chapter is as follows: the goals of this research are set out and clarified in section 1.1. The general model-based testing process is set out in section 1.2. Some basic concepts from first order logic are described in section 1.3, used as a framework for dealing with data in the symbolic models used in ATM. These models are then described in section 1.4. Section 1.5 describes the graph grammar formalism. GROOVE and ATM are described in section 1.6. Section 1.7 describes the use of variables and algebras in graph grammars, as used in GROOVE.

[1] http://sourceforge.net/projects/groove/
[2] http://www.axini.nl/

1.1 Research Goals

The research goals are split into a design and validation component:

1. **Design**: Design and implement a system using ATM and GROOVE which performs model-based testing on graph grammars.
2. **Validation**: Validate the design and implementation using case studies and performance measurements.

The result of the design goal is one system called the GROOVE-Axini Testing System (GRATiS). The validation goal uses case-studies with existing specifications from systems tested by Axini. Each case-study has a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the models and the test processes are compared as part of the validation.

1.2 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in Figure 1.1. The specification of a system, given as a model, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and the responses. It gives a 'pass' or 'fail' verdict whether the SUT conforms to the specification or not respectively.

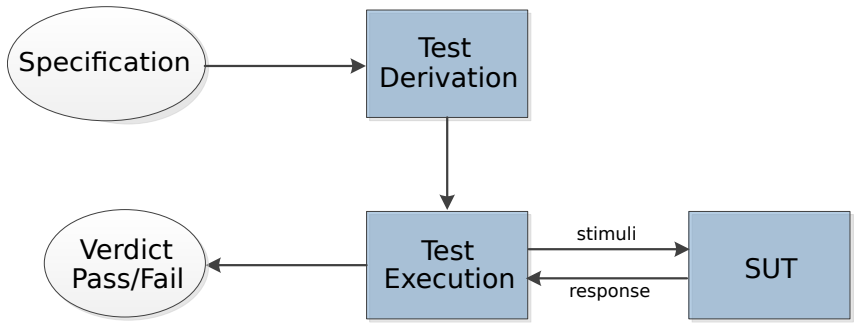


Figure 1.1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on the fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 1.2. A tool for on-the-fly testing is TorX [24], which integrates automatic test generation, test execution, and test analysis. A version of this tool written in Java under continuous development is JTorX [2].

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data. First, previous work on model-based testing is given. Then, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Finally, the notion of *coverage* is explained.

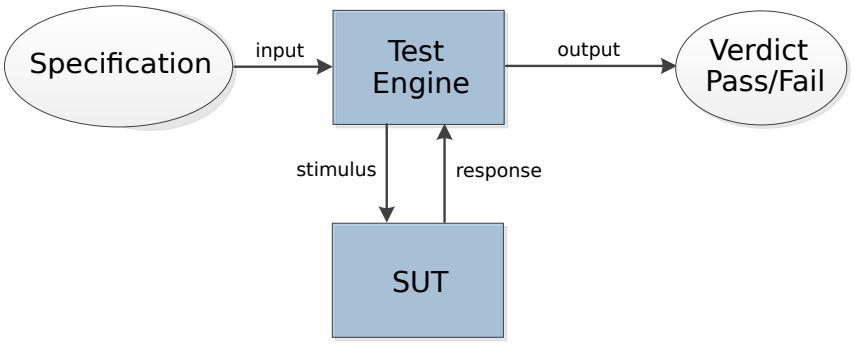


Figure 1.2: A general 'on-the-fly' model-based testing setup

1.2.1 Previous work

Formal testing theory was introduced by De Nicola et al. [17]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [25] and an algorithm for deriving a sound and exhaustive test suite from a specification in [26]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [12].

1.2.2 Labelled Transition Systems

A labelled transition system is a structure consisting of states with labelled transitions between them.

Definition 1.2.1. A labelled transition system is a 4-tuple $\langle Q, L, T, q0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

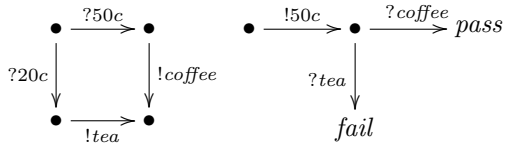
1.2.3 Input-Output Transition Systems

A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [26]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $t \in T$, where $T = \{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test.

An example of such an IOTS is shown in Figure 1.3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a question mark, the

outputs are preceded by an exclamation mark. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states. A test case for the coffee machine is given in Figure 1.3b. The test case shows that when an input of '50c' is done, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 1.3: The specification of a coffee machine and a test case as an IOTS

1.2.4 Coverage

The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the SUT is tested. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the model [10, 16, 6].

As an example, let us calculate the coverage metrics of the IOTS test case example in 1.3b. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [27]. These statisics are therefore useful metrics for communicating how much of a system is tested.

1.3 First Order Logic

Some basic concepts from first order logic are described here, used in the definitions of section 1.4. For a general introduction into logic we refer to [8].

Let $\mathcal{V}$ be a set of *variables*. *Terms* over V, denoted $\mathcal{T}(V)$, are built from a set of function symbols F and variables $V \subset \mathcal{V}$. Each $f \in F$ has a corresponding arity $n \in \mathbb{N}$. if $n = 0$ we call f a constant. We write $var(t)$ to denote the set of variables appearing in a term t. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms.

A *term-mapping* is a function $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{V})$. For sets V, W with $V \cup W \subset \mathcal{V}$, we write $\mathcal{T}(W)^X$ for the set of term-mappings that assign to each variable $v \in V$ a term $t \in \mathcal{T}(W)$, and to each variable $v \notin V$ the term v.

A *valuation* ν is a function $\nu : \mathcal{V} \mapsto \mathcal{U}$, where $\mathcal{U}$ is a non-empty set called a *universe*. An example of a universe is $\mathbb{N}$, the non-zero integers. Additionally, $\nu : n - tuple : x \mapsto n - tuple : y$, maps the values of tuple x to the values of tuple y.

An *evaluation* of $\mathcal{T}(\mathcal{V})$ is given by $\epsilon : (\mathcal{T}(\mathcal{V}), \nu : \mathcal{V}) \mapsto \mathcal{U}$. An evaluation $\epsilon : \mathcal{T}(\mathcal{V}), \nu_{before} : \mathcal{V})$ of the terms in a term-mapping results in a set $X \subseteq \mathcal{U}$. A new valuation can then be constructed by: $\nu_{after} : \mathcal{V} \mapsto X$.

1.4 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. These systems are used in practice in ATM and will therefore be part of GRATiS. First, previous work on STSs is given. The definition of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the transformation of an STS to an LTS is explained and illustrated by an example. This transformation is useful when comparing STSs to systems that are not STSs. Finally, different coverage metrics on STSs are explained.

1.4.1 Previous work

STSs are introduced by Frantzen et al. [9]. This paper includes a detailed definition, on which the definition in section 1.4.2 is based. The authors also give a sound and complete test derivation algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [22]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [9]. However, the definition of IOSTSs we will use in this report is based on the STSs by [9]. A tool that generates tests based on symbolic specification is the STG tool, described in Clarke et al. [4].

1.4.2 Definition

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings*. A guard is a term t, where any evaluation on t with any valuation results in a value from $\mathbb{B} = \{true, false\}$. Such a term is denoted by $\mathcal{F}(\mathcal{V})$. The guard disallows using the switch relation when the evaluation of the term results in $false$. When the evaluation results in $true$, the switch relation of the guard is *enabled*. An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the evaluation of the term they map to.

Definition 1.4.1. A Symbolic Transition System is a tuple $\langle L, l_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, \rightarrow \rangle$, where:

- L is a finite set of locations and $l_0 \in L$ is the initial location.
- $\mathcal{L}$ is a finite set of location variables.
- $\imath$ is a term-mapping $\mathcal{L} \rightarrow \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.
- $\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{L}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.

- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{F}(\mathcal{V} \cup \mathcal{I}) \times \mathcal{L} \mapsto \mathcal{T}(\mathcal{L} \cup \mathcal{I}) \times L$, is the switch relation. We write $l \xrightarrow{\lambda,\phi,\rho} l'$ instead of $(l, \lambda, \phi, \rho, l') \in \rightarrow$, where ϕ is referred to as the guard and ρ as the update mapping. We require $var(\phi) \cup var(\rho) \subseteq \mathcal{V} \cup param(\lambda)$, where var is the collection of the variables used in the given guard or update mapping. We define $l \rightarrow \subset \rightarrow$ to be the outgoing switch relations from location l.

1.4.3 Input-Output Symbolic Transition Systems

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type $t \in T$, where $T = \{input, output\}$. As with IOSTSs, each gate is preceded by a '?' or '!' to indicate whether it is an input or an output respectively.

1.4.4 Example

In Figure 1.4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The defining tuple of the IOSTS is:

$\langle \{throw, move\}, throw, \{T, P, D\}, \{T \mapsto 0, P \mapsto [0,2], D \mapsto 0\}, \{d, p, l\}, \{?throw, !move\},$
$\{throw \xrightarrow{?throw, 1<=d<=6, D \mapsto d} move, move \xrightarrow{!move, T=p \wedge l=(P[p]+D)\%4, P[p] \mapsto l, T \mapsto p\%2} throw\}\rangle$

The variables T, P and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $!move$ has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate $?throw$ has $param = \langle d \rangle$ symbolizing which number is thrown by the die. The switch relation with gate $?throw$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $!move$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 1.4 the gates, guards and updates are separated by pipe symbols '|' respectively.

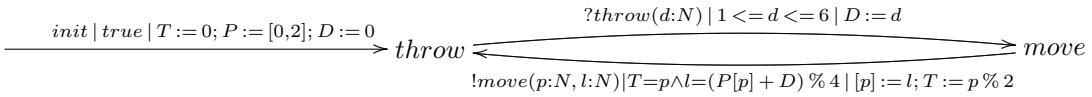


Figure 1.4: The STS of a board game example

1.4.5 STS to LTS transformation

Consider an STS S and its transformation LTS L. The following defines a mapping between $\mathcal{L}$, $\mathcal{V}$, $\mathcal{I}$ and $\rightarrow$ of S to the states Q and transitions T of L.

Definition 1.4.2.

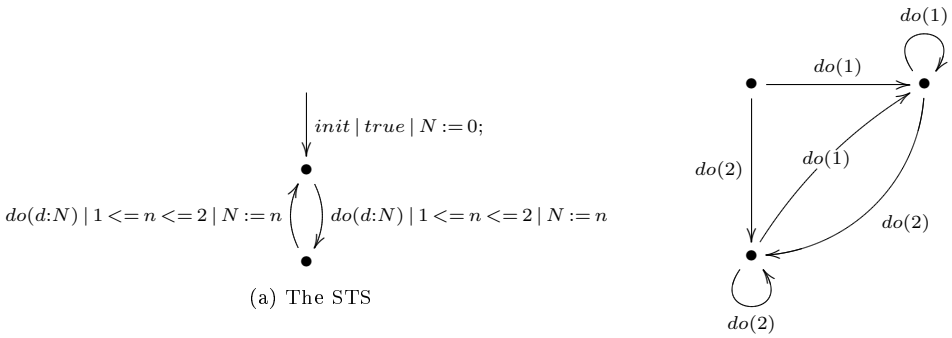
$\mu_l : (l \in L, \nu : \mathcal{V}) \mapsto q \in Q$

$\mu_r : (r \in \rightarrow, \nu : \mathcal{I}) \mapsto t \in T$

Finding the topology of L is the next step of the transformation. For a switch relation r from location A to location B, a valuation of the location variables ν_l and interaction variables ν_i, $\mu_l : (A, \nu_l)$ maps to a state q, where q is the source state of a transition t, if the result of the evaluation $\epsilon : (\phi \text{ of } r, \nu_l \cup \nu_i)$ is true. ν_{l-new} is the new valuation of the location variables

constructed by the evaluation of ρ of r. Then, the target state q' of t is the state mapped by $\mu_l : (B, \nu_{l-new})$. The label of t is a textual representation of λ of r and ν_i. Applying this rule for the topology to all locations, switch relations and concrete values for the variables, results in L. The start state $q0$ of L is the state mapped by $\mu_l : (l_0, \imath)$. All states not reachable from $q0$ are removed from L. When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ and the number of locations in an STS is considered to be finite, the transformation is always possible to an LTS with finite number of states.

An example of this transformation is shown in Figure 1.5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The transformation of a switch relation and concrete values to a transition is also called *instantiating* the switch relation. Another term we will use for a switch relation with a set of concrete data values is an *instantiated switch relation*.



(a) The STS

(b) The LTS

Figure 1.5: An example of a transformation of an STS to an LTS

1.4.6 Coverage

The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested in the test run. Measuring state and transition coverage of an STS is possible using the LTS resulting from the STS transformation. However, this metric is not always useful, because the number of states and transitions in the LTS depend on the number of unique combinations of concrete values of the variables in the STS. This is potentially very large. For example, when the guards of the switch relations in Figure 1.5a are removed, the transformation leads to an LTS with a state and transition for each possible value of an integer. It is often unfeasable to test every data value in the STS. The most interesting data values to test can be found by *boundary-value analysis* and *equivalence partitioning*. For an explanation of these terms we refer to [15]. Boundary-value analysis was found to be most effective by Reid [19] in fault detection.

Data coverage expresses the percentage of data tested in the test run, considering data to be similar if located in the same partition and a better representative of the partition if located close to the partition boundary. These properties of the tested data affect the data coverage percentage.

1.5 Graph Grammars

A *graph grammar* is composed of a start graph and a set of transformation rules. The start graph describes the system in its initial state. The transformation rules describe what changes are made

to the graph, resulting in a new graph which describes the system in its new state. The definition is as follows.

Definition 1.5.1. A graph grammar is a tuple $\langle G, R \rangle$, where:

- G is the start graph
- R is a set of graph transformation rules

The rest of this section is ordered as follows: first, graphs and graph morphisms are explained. This is then used to explain graph transformation rules, followed by the definition of a graph grammar. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a graph grammar and a GTS is then given. Finally, a method for transforming a GTS to an STS is given. For a more detailed overview of graph grammars, we refer to [20, 7, 1].

1.5.1 Graphs & morphisms

Definition 1.5.2. A graph is a tuple $\langle L, N, E \rangle$, where:

- L is a set of labels
- N is a set of nodes, where each $n \in N$ has a label $l \in L$
- E is a set of edges, where each $e \in E$ has a label $l \in L$ and nodes $source, target \in N$

A graph H has an *occurrence* in a graph G, denoted by $H \to G$, if there is a mapping from the nodes and the edges of H to the nodes and the edges of G respectively. Such a mapping is called a *morphism*. An element e in graph H is then said to have an *image* in graph G and e is a *pre-image* of the image. A graph H has a partial morphism to a graph G if there are elements in H without an image in G.

1.5.2 Graph transformation rules

Definition 1.5.3. A transformation rule is a tuple $\langle LHS, NAC, RHS, M \rangle$, where:

- LHS is a graph representing the left-hand side of the rule
- NAC is a set of graphs representing the negative application conditions
- RHS is a graph representing the right-hand side of the rule
- M_{RHS} is a partial morphism of LHS to RHS
- M_{NAC} are partial morphisms of LHS to each $n \in NAC$

A rule R is applicable on a graph G if its LHS has an occurrence in G and $\nexists n \in NAC$ such that n has an occurence in G and $\forall e \in LHS$, if e has an image i in n, i.e. $(e \mapsto i) \in M_{NAC}$, and an image j in G, then j should be an image of i. This 'applicability' as defined here is refered to as a rule *match*. After the rule match is applied to the graph, all elements in LHS not part of M_{RHS}, i.e. they do not have an image in RHS, are removed from G and all elements in RHS not part of M_{RHS}, i.e. they do not have a pre-image in LHS, are added to G.

1.5.3 Graph Transition Systems

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a graph grammar can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of *graph states* connected by *rule transitions*.

Definition 1.5.4. A graph transition system is an 8-tuple $\langle S, L, T, G, R, M_G, M_R, s0 \rangle$, where:

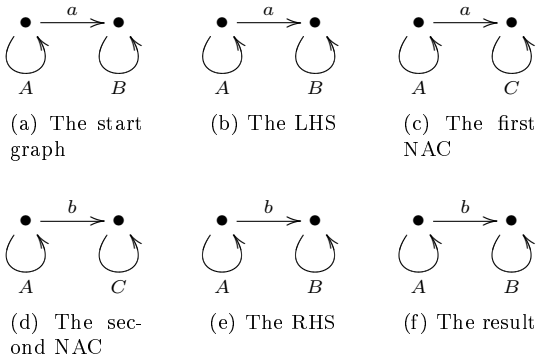
- S is a finite, non-empty set of graph states
- L is a finite set of labels.
- $T \in S \times (L \cup \{\tau\}) \times S$, with $\tau \notin L$, is the rule transition relation.
- G is a set of graphs.
- R is a set of rules.
- M_G is a mapping $\forall s \in S.s \mapsto g \in G \wedge \nexists s' \in S.s \neq s' \wedge s' \mapsto g \in M_G$
- M_R is a mapping $\forall t \in T.t \mapsto r \in R \wedge \nexists t' \in T.t \neq t' \wedge t' \mapsto r \in M_R$
- $s0 \in s$ is the initial graph state.

We write $s \xrightarrow{\mu} s'$ if there is a rule transition labelled μ from state s to state s', i.e., $(s, \mu, s') \in T$.

These systems are very similar to LTSs. A GTS can be transformed to an LTS by omitting the graphs, rules and mappings.

1.5.4 Example

Figure 1.6 shows an example of the start graph and one rule of a graph grammar. M_{RHS} maps the A and B nodes in *LHS* to the A and B nodes in *RHS* respectively. M_{NAC} maps the A node in *LHS* to the A node in both graphs in *NAC*. The a-edge in *LHS* is mapped to the a-edge in the first *NAC*. The *LHS* of the rule has an occurrence in the start graph, as the A and B nodes connected by the a-edge exist in both graphs. None of the graphs in the *NAC* have an occurrence in the start graph, because the C node does not exist in the start graph. The new graph after applying the rule is in Figure 1.6f.



(a) The start graph (b) The LHS (c) The first NAC

(d) The second NAC (e) The RHS (f) The result

Figure 1.6: An example of a graph grammar

1.5.5 Input-Output Graph Grammars

In order to specify stimuli and responses with graph grammars, a definition is given for an *Input-Output Graph Grammar* (IOGG). Concretely, the IOGG places input and output labels on its rule transitions. Following the definition from IOLTSs, each rule transition label $l \in L$ has a type $t \in T$, where $T = \{input, output\}$. Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS).

1.6 Tooling

1.6.1 ATM

ATM is a web-based model-based testing application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 1.7. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 1.2.

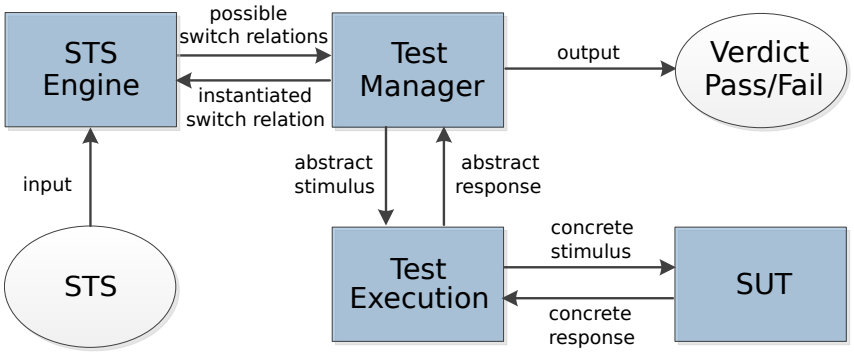


Figure 1.7: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and data values. It passes the possible switch relations from the current location to the Test Manager.
2. The Test Manager chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation and passed back to the STS Engine, which updates its current location and data values. The communication between these two components is done by method calls.
3. The gate of the instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label 'connect?' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.
4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label 'ok!'. The Test Manager is notified with this abstract response.
5. The Test Manager translates the abstract response to an instantiated switch relation and updates the STS Engine. If this is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.

1.6.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [21]. It has been applied to several case studies, such as model transformations and security and leader election protocols [5].

The architecture of the GROOVE tool is shown graphically in Figure 1.8. A graph grammar is given as input to the Rule Applier component, which determines the possible rule transitions. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the possible rule transitions and respond with the chosen rule transition (based on the exploration strategy or the user input). The Exploration Strategy can do an exhaustive search, resulting in a GTS. The graph states and rule transitions in this GTS can then be inspected using the GUI.

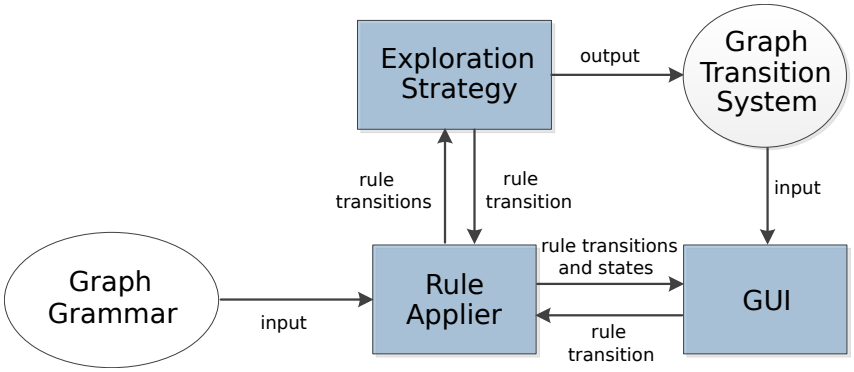


Figure 1.8: The GROOVE Tool

1.7 Variables in Graph Grammars

The definition of graph grammars, as given in section 1.5, does not support the use of variables and algebraic expressions. This makes modelling a software system with integers difficult. In this section, we extend our definition of graph grammars to include 'variables' in a graph. Also, a sketch is given for *graph algebras*.

Definition 1.7.1. A graph grammar is a tuple $\langle G, R, V \rangle$, where:

- G is the start graph
- R is a set of graph transformation rules
- V is a set of nodes which are *variable nodes*

Also, the definition of GTSs is extended:

Definition 1.7.2.

- L is a finite set of labels. For each label $l \in L$, the arity of l, denoted $arity(l)$, is a natural number. The parameters of l, denoted $param(l)$, is a tuple of length $arity(l)$ of variables.
- $T \in S \times (L \cup \{\tau\}) \times S$, with $\tau \notin L$, is the rule transition relation. The parameters of a transition $t \in T$ with label $l \in L$, denoted $param(t)$, is a tuple of length $arity(l)$ of constants, such that $\Sigma : param(l) \mapsto param(t)$ is the valuation of the variables of the label.

The concrete implementation of this can differ. For example, a variable node can be restricted to one self edge giving the name and an edge labeled 'value' to a node with one self-edge, representing

the value of the variable. We will use the term *value node* to indicate this node. Figure 1.9 shows an example of a variable node with name 'x' and a value node with the value '25'.

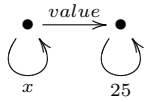


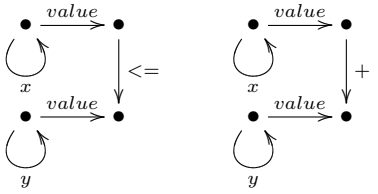
Figure 1.9: An example of a variable in a graph

We will use this example of a variable/value node pair in chapter 2.

1.7.1 Graph algebras

With the existence of variables in graphs, algebraic expressions over these variables is a possibility. Again, the implementation of this can differ. Next is an example of a possible implementation. A formal definition of graph algebras is not given, as the use of algebras in this report is restricted to the examples.

Figure 1.10 shows the LHS and RHS of a rule. M_{RHS} maps the variable nodes 'x' and 'y' and their value nodes in the LHS to the RHS. The '<=' and '+' labels have a special meaning here. The edge with the '<=' label is omitted from the rule matching, however, the rule will not match if the value of 'x' is larger than the value of 'y'. The result of this rule is not the creation of an edge from the value node of 'x' to the value node of 'y', but the creation of a self-edge on the value node 'x' with label equal to the value of 'x' plus the value of 'y'.



(a) The LHS (b) The RHS

Figure 1.10: An example of algebraic expressions in a rule

We will use this example of algebraic expressions over variable nodes in chapter 2.

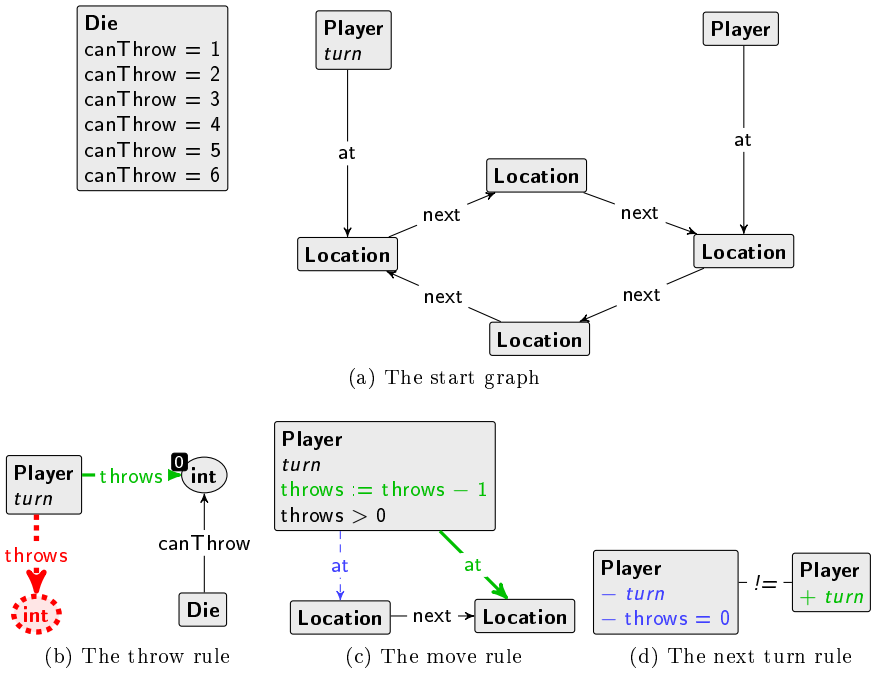
1.7.2 Graph grammars in GROOVE

The graph grammars in GROOVE give support to variables and graph algebras. In the following example, a graph grammar is shown as modelled in GROOVE.

The running example from Figure 1.4 is displayed as a graph grammar, as visualized in GROOVE, in Figure 1.11. The *LHS*, *RHS* and *NAC* of a rule in GROOVE are visualized together in one graph. Figures 1.11b, 1.11c and 1.11d show three rules. Figure 1.11a shows the start graph of the system.

The colors on the nodes and edges in the rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.



(a) The start graph

(b) The throw rule

(c) The move rule

(d) The next turn rule

Figure 1.11: The graph grammar of the board game example in Figure 1.4

3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The rules can be described as follows:

1. 1.11b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 1.11c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 1.11d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The strings on the nodes are a short-hand notation. The bold strings, **Die**, **Player**, **Location** and **int** indicate the *type* of the node. Nodes with a type starting with a lower case letter, such as **int**, are variable nodes. The italic string *turn*, is a representation of a self-edge with label *turn*. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node and in the *RHS* as a self-edge of the right **Player** node. In the same rule, the *throws* edge from the left **Player** node to an integer node only exists in the *LHS*.

The assignments on the **Die** node are representations of edges labelled 'canThrow' to variable nodes. The six variable nodes are of the type integer and each have an initial value of one to six. The throws value assignment ($:=$) in the move rule is a shorthand for two edges: one edge in the *LHS* with label *throws* from the **Player** node to an integer node with value i and another edge in the *RHS* with label *throws* from the **Player** node to an integer node with value $i - 1$.

The 'throws $>$ 0' expression is an example of the use of algebra on variable nodes. In this case, the expression in boolean algebra must evaluate to true for the rule to match the graph.

The number '0' in the top left of the **int** node in the throw rule indicates that this integer is the first parameter in $param(l)$, where l is the label on the rule transition created by applying the throws rule.

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 1.12 shows the IOGTS of one *?throws* rule application on the start graph. Note that the *?throws* is an input, as indicated by the '?'. State s_1 is a representation of the graph in Figure 1.11a. Figure 1.13 shows the graph represented by s_2.

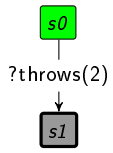


Figure 1.12: The GTS after one rule application on the board game example in Figure 1.11

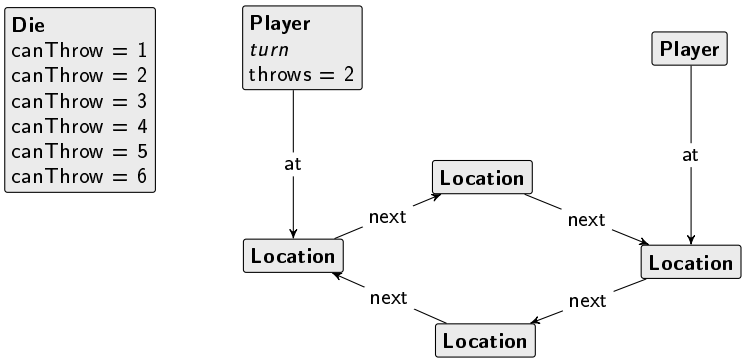


Figure 1.13: The graph of state $s2$ in Figure 1.12

Chapter 2

Design

This chapter covers the design issues of GRATiS. GRATiS transforms a graph grammar to an STS and starts the model-based testing on that STS. This design choice was made, because STSs are practical for model-based testing; variables give support for modelling data values in systems and using STSs supports the separation of state/transition coverage in location/switch relation coverage and data coverage.

Section 2.1 gives a formal approach of transforming a graph grammar to an STS. Section 2.2 gives possible optimizations. Section 2.3 shows the design of GRATiS and elaborates on the choices made. Section 2.4 gives implementation problems and solutions on GROOVE and ATM.

2.1 Graph Grammar to STS Transformation Rules

2.1.1 Graph states transformation

Graph states are a representation of the state of a system. In an STS, this representation is done by the locations and the location variables. In order to obtain an STS from a graph grammar, the explored graph states are transformed to locations and location variables.

The locations L of the transformation are obtained by abstracting the variable nodes in the graph states. For the example given of graphs with variable nodes specifically, the value nodes are removed from the graph. The abstracted graph states are considered to be the same location in the transformed STS when they are isomorphic. The location variables $\mathcal{L}$ are identified by the variable nodes V. The value nodes in the start graph give the initialization $\imath$ of the STS.

Figure 2.1 shows the abstraction of the graph in Figure 1.9. Assuming this is the graph of a start graph state, the transformed STS is as follows: $L = \{l_0\}, \mathcal{L} = \{x\}, \imath = \{x \mapsto 25\}$, where the start location l_0 is represented by the abstraction.

Figure 2.1: An example of a graph abstraction

2.1.2 Rule transitions transformation

Rule transitions are a representation of the switch from one system state to another. In STSs, this representation is done by the switch relations. In order to obtain an STS from a graph grammar, the explored rule transitions are transformed to the switch relations.

2.1.2.1 Rule LHSs

When the LHS of a rule has an occurence in the graph of a graph state and the NACs of the rule do not, a rule transition exists. This also indicates the presence of a switch relation $r \in \rightarrow$ in the STS. The label on the rule transition is transformed to a gate $\lambda \in \Lambda$ in the STS. The source location of the switch relation is the transformation of the source graph state of the rule transition and like-wise with the target location. This means that multiple rule transitions can transform to the same switch relation.

When a variable node is an image of a node in the LHS, this indicates a constraint on the variable represented by that node. In some graph states, the value node of the variable node will be an image of a node in the LHS, whereas in other graph states it will not. When the abstractions of these graph states are isomorphic, i.e. transform to the same location, an outgoing switch relation represented by the rule transitions is present. However, since there are graph states where the switch relation should not be enabled, a guard ϕ must be specified on this switch relation. This guard specifies the possible values the location variable represented by the variable node may have.

Figure 2.2 shows the LHS of a rule. This graph has an occurrence in a graph where the variable 'x' has the value '25'. In the graph grammar to STS transformation, this LHS will cause a 'x = 25' guard to be set on switch relations.

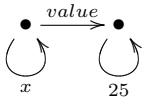


Figure 2.2: The LHS of a rule

2.1.2.2 Rule NACs

The absence of occurences of the NACs of a rule indicates the existence of a rule transition. However, as with the LHS, when the value node of a variable node is the image of a node in a NAC, a constraint is specified on that value. With the same reasoning as in section 2.1.2.1, this constraint must be included in the guard ϕ. The guard then also specifies the possible values the location variable represented by the variable node may not have.

Figure 2.3 shows a NAC of a rule. This graph has an occurrence in a graph where the variable 'x' has the value '10'. In the graph grammar to STS transformation, this NAC will cause a 'x != 10' guard to be set on switch relations.

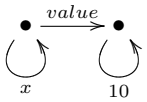


Figure 2.3: A NAC of a rule

2.1.2.3 Rule RHSs

The RHS of a rule gives the target graph state of a rule transition. This indicates the target location of the switch relation represented by the rule transition. However, when a variable node is the image of a node in the RHS and the value node of the image has no pre-image in the RHS, a change of the value of the variable has occurred. This change can be made into an update mapping ρ in the transformed STS; the location variable represented by the variable node is mapped to the value specified by the value node in the RHS. Note that we assume the existence of such a node in the RHS.

Figure 2.4 shows the RHS of a rule. In the graph grammar to STS transformation, this RHS will add $x \mapsto 5$ to the update mapping of a switch relation.

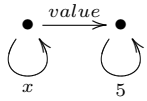


Figure 2.4: The RHS of a rule

Figures 1.9, 2.2, 2.3 and 2.4 are a graph grammar with one rule, which we will name 'rule' for the purpose of this example. The complete transformation according to the rules in this section is shown in Figure 2.5.In this particular case, the LHS and the NAC are mutually exclusive, so the guard could be optimized. However, it still demonstrates the principle of the transformation rules.

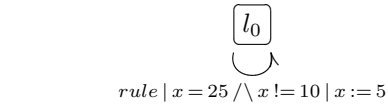


Figure 2.5: The STS transformation of a simple graph grammar

2.2 Symbolic Graph Grammar Exploration

In the previous section, rules are given for transforming graph states and rule transitions to an STS. However, completely exploring the GTS is often unfeasable, as GTSs, like LTSs, have states for each concrete data value. This section describes an exploration technique to transform a graph grammar to an STS without having to explore the entire GTS.

2.2.1 Partial matching

When a rule match of a rule r exists on a graph G, if the *LHS* of r is allowed to only have a partial morphism in a graph G, this is called a *partial match* on G. Note that according to the definition given in section 1.5.2, r is applicable on G when a *NAC* has no occurrence in G, where an image of an element in the *NAC* should also be an image of an element in the *LHS*. This means that a rule which does not have a match on a graph, because of its *NAC*, may still have a partial match on the graph. For example, the *LHS* in Figure 2.2 has a partial morphism on the graph in Figure 1.9, if the value in the latter is '10' instead of '25'. The rule with this *LHS* and the *NAC* in Figure 2.3 has a partial match on this graph, as the edge labelled '10' has no image in the *LHS*.

When a rule has a partial match, excluding the value nodes, on the graph of a graph state s, it indicates the existence of an outgoing switch relation from the location represented by the abstraction of s in the STS transformation. The guard and update mapping can be constructed from the rule, by inspecting the variable and value nodes in the *LHS*, *NAC* and *RHS*. Whether this switch relation is ever enabled, depends on whether the graph states on which the rule has a full match are reachable from the start graph state. When exploring a GTS with this partial matching technique, graph states can be omitted from the exploration when their abstraction has already been encountered. This ensures that for each switch relation in the STS transformation, only one rule transition is explored.

The rule in Figure 1.10 in section 1.7 leads to a new graph state for each possible value of 'x'. A graph grammar with this rule and the initial graph state in Figure 2.6 leads to a GTS as shown in Figure 2.7, where s_0, s_1 and s_2 are representations of the start graph state, the graph state after one rule application and the graph state after two rule applications respectively. The abstractions of all three graph states are isomorphic, because only the value of the variable 'x' changes. The transformation of this graph grammar to an STS using partial matching is as follows: The start graph state is transformed to the start location l_0. The value nodes are not included in the partial match, so the rule transition leads to the same graph state. This partial rule match is transformed to a switch relation with gate *add* from l_0 to l_0. By inspecting the LHS, the guard on r can be set to $x <= y$. By inspecting the RHS, the update mapping on r can be set to $x \mapsto x + y$. The result is shown in Figure 2.8.

This exploration technique with partial matching can potentially lead to an infinitely continuing exploration. An example of this is given using the same start graph and rule as in Figures 2.6 and 1.10. However, the RHS of the rule is changed to the one in Figure 2.9. A new graph state is found with each application of the rule. With the partial matching system, this means that the STS from the transformation will expand infinitely. With the normal matching system, this is not the case, as the rule can only be applied two times and then the rule does not match anymore. One

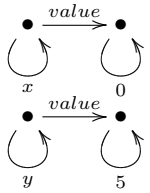


Figure 2.6: A start graph state

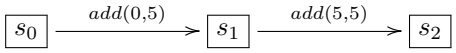


Figure 2.7: The GTS resulting from Figures 2.6 and 1.10

solution is to set a modelling constraint on GRATiS. This constraint disallows the use of a rule that causes an infinite expansion of graph states with the partial matching technique. Another solution is to allow user input for the maximum depth of the exploration. However, the coverage statistics for such models are often incorrect, as many locations are still unreachable. Section 2.2.2 provides a solution to this problem.

2.2.2 Reachability

With each new location in the transformed STS the question arises whether that location is reachable from the start location. The MSc thesis of Sietsma [23] gives an algorithm for checking whether a location is reachable. This algorithm works on a specific path, starting from the start location and ending in the location to check. However, with the presence of loops, the number of paths is infinite. A location deemed as unreachable can therefore still be reachable, following a different path. Still, this factor can be included in the coverage statistics: the number of reachability checks to each unreachable location can be reported. This allows the tester to determine whether additional testing to these 'unreachable' locations is necessary.

2.3 GRATiS Design

2.3.1 Offline vs. on-the-fly model exploration

The exploration of the graph grammar can be done in two ways: *on the fly*, rule transitions are explored only when chosen by the Test Manager, or *offline*, the graph grammar is transformed to an STS and sent to the Test Manager.

On-the-fly model exploration does not encounter the reachability problems discussed in the previous section. However, coverage statistics cannot be calculated on GRATiS, if the model exploration is done on the fly. The number of states/locations and transitions/switch relations the model has when completely explored are not known, so a percentage cannot be derived. As coverage statistics are an important metric, the offline model exploration is chosen for GRATiS.

2.3.2 Tool architecture

GRATiS uses GROOVE as a replacement of the STS in ATM. Figure 2.10 shows this graphically. GROOVE has several exploration strategies for exploring a graph grammar. A new strategy

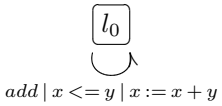


Figure 2.8: The result of the graph grammar to STS transformation

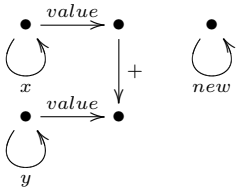


Figure 2.9: The RHS of a rule leading to infinite continuing exploration

is added, the *symbolic exploration strategy*. This strategy transforms the graph states and rule transitions found to an STS, following the transformation rules in section 2.1.

The interface on the GROOVE side is implemented by means of a *remote exploration strategy*. This strategy uses the symbolic exploration strategy to derive an STS and sends an HTTP POST request with the STS formatted as JSON to the ATM interface.

The ATM interface receives the HTTP request, parses the JSON to an STS, starts the STS Engine with the STS and initiates the test run.

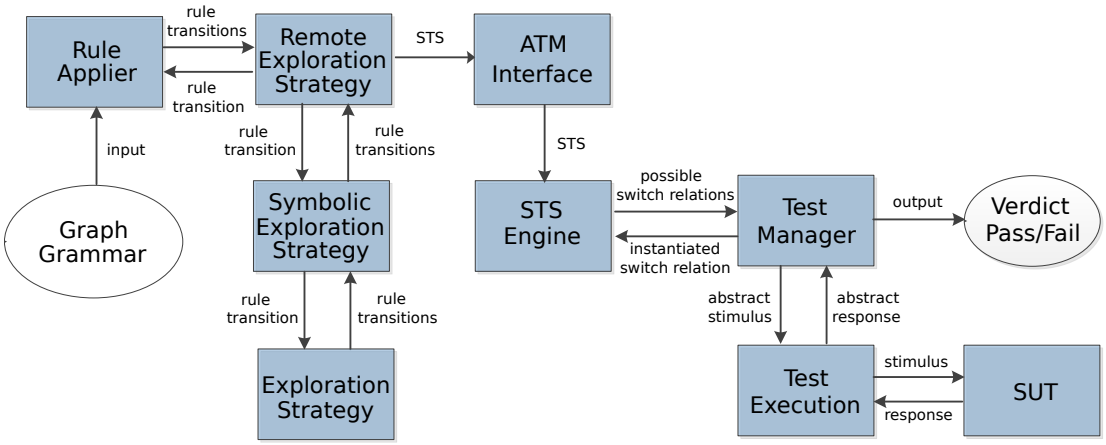


Figure 2.10: The GRATiS design: replacing the STS with GROOVE

2.4 Implementation

This section features several implementation issues of GRATiS.

2.4.1 Point algebra

In Figure 1.11 the support GROOVE gives to algebraic expressions can be seen. Different algebra families can be set for a graph grammar, such as the *point algebra*. It has an interesting property:

the value of any variable or expression is the same for each variable type. Therefore, the value nodes in each graph state are the same. In the transformation rules, two graph states are considered to transform to the same location when they are isomorphic, omitting the values of the variables. Thus, exploring a graph grammar using the point algebra, produces the same result as the partial matching scheme in section 2.2.1.

The boolean expressions in the rules are also affected by the point algebra: the result of a boolean expression is always true as this is the default value of a boolean in the point algebra. Therefore, the effect of using this algebra family is that all constraints on the values of variable nodes expressed by a rule system are ignored. There is one exception, shown in Figure 2.11. This is a shorthand for the construction in Figure 2.12. The values are directly compared, therefore this will always prevent a rule match when using a point algebra, because the values will always be equal. The problem is solved by setting a modelling constraint on GRATiS, disallowing the use of the not-equals construction in Figure 2.11. Instead, the construction in Figure 2.12 should be used.

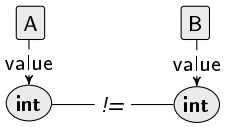


Figure 2.11: A not equals expression in GROOVE

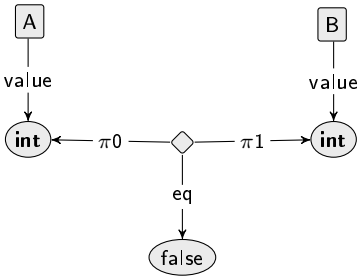


Figure 2.12: Another not equals expression in GROOVE

2.4.2 Control program

Modelling a random value in a graph grammar is difficult. For example, to model a die, all possible values of the die must be included in the model, as in Figure 1.11. Then, a rule transition can be created for each of these values. The random 'choice' is now a non-deterministic one. However, when using a point algebra, this can be done easier. The rule in Figure 2.13 represents the possible values of the die with lower- and upper-bounds. The Die node is still required to produce a match, because the rule systems in GROOVE cannot match an 'unknown' variable node. A variable node is unknown when it has no value and no connecting edges. Note that the edges expressing the lower- and upper-bound are not part of the match and therefore not considered as connecting edges.

The Die node in the rule is visually intuitive, as it shows that the random value is produced from a die. However, in the start graph, the Die node must be present with at least one value. This is not an intuitive part of the model and we would preferably omit this. A solution is using a *control program*. The rule in Figure 2.14a shows an **int:** node, with '?0' in the top left, which obtains its value from an external program. An example of such a program is in Figure 2.14b. The program tries, as long as possible ('alap'), to set the value of the **int:** node in the 'throws' rule to '1'. This node still matches other values, when using a point algebra.

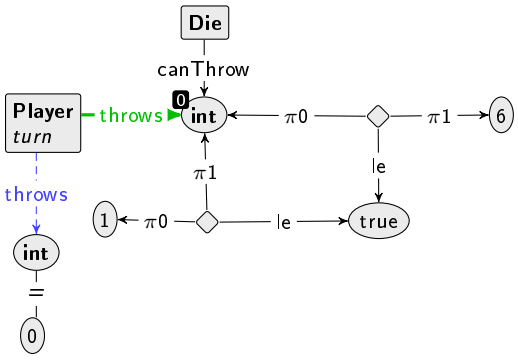
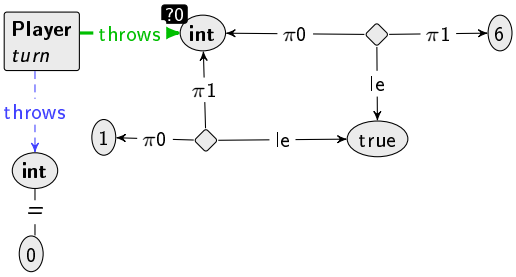


Figure 2.13: A rule in GROOVE



(a) The rule with control node

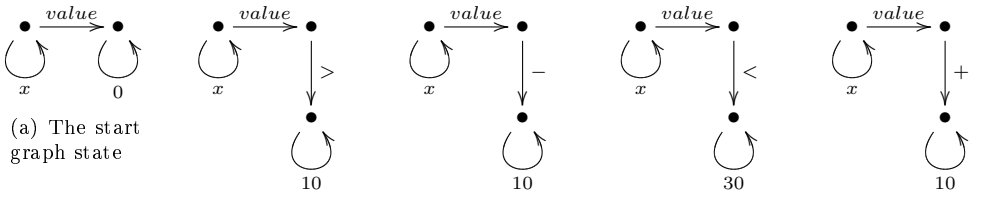
alap { throws(1); }

(b) The control program

Figure 2.14: A rule with control node and program in GROOVE

2.4.3 Rule priority

From a graph state, there can be several outgoing rule transitions. However, GROOVE can set different priority levels on rules. A rule transition with a higher priority rule is explored before rule transitions with lower priority rules. Consider the graph grammar in Figure 2.15. The 'add' rule produces a rule transition to a graph state, where the 'sub' rule produces a rule transition back to the start graph state. The 'sub' rule is not applicable in the start graph state, because it has a lower priority than the 'add' rule.



(a) The start graph state

(b) The LHS of rule 'sub' with priority 1

(c) The RHS of rule 'sub' with priority 1

(d) The LHS of rule 'add' with priority 2

(e) The RHS of rule 'add' with priority 2

Figure 2.15: A control node and program in GROOVE

The abstractions of these graph states are isomorphic, so they represent the same location. The STS of transforming this graph grammar is in Figure 2.16, with $\imath = \{x \mapsto 25\}$. This STS is wrong, because the 'sub' switch relation can be taken from the start.

The solution is shown in Figure 2.17. The negated guard of the 'add' switch relation is added to

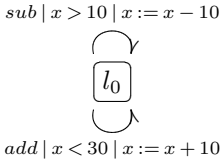


Figure 2.16: A wrong STS transformation of the graph grammar in Figure 2.15

the 'sub' switch relation. The optimized guard for this switch relation is 'x >= 30' of course, but this shows the main principle: for each outgoing switch relation, the negated guard of all switch relations represented by higher priority rules must be added to the guard. So, the 'x < 30' guard is negated to '!(x < 30)' and added to yield the 'x > 10 && !(x < 30)' guard. Note that if the 'add' switch relation had no guard, it would be applicable on all graph states with isomorphic abstractions. Therefore, the 'sub' switch relation would not exist, because the 'add' rule is always applicable whenever the 'sub' rule also is.

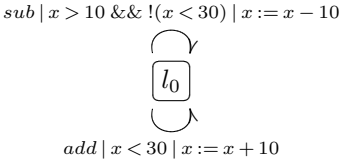


Figure 2.17: A correct STS transformation of the graph grammar in Figure 2.15

Chapter 3

Validation

This chapter covers the validation of the design. The validation is done through case-studies, reported in section 3.1. Measurements on models and the performance are reported in section 3.2.

3.1 Case Studies

Three case studies with GRATiS have been done. They are all real-life systems Axini has worked on:

- a self-scan register
- a navigation system
- a health-care system

The self-scan register is a machine that automates the purchase of products at a supermarket. A customer can put his products on a conveyer belt and the system automatically calculates the price of the products. Then the customer pays and gets a receipt. The navigation system is a GPS device with a route planner. It allows a user to enter a destination and the system plans the correct route accordingly from the location of the user. The health-care system is a medical device.

A graph grammar and an STS are created for each system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process are compared as part of the validation. Design issues, parts of the models and observations are noted in sections 3.1.1 through 3.1.3. The objective measurments are reported in section 3.2.

3.1.1 Self-scan register

Put design issues here. Parts of model if enough room?

3.1.2 Navigation system

Put design issues here. Parts of model if enough room?

3.1.3 Health-care system

Put design issues here. Parts of model if enough room?

3.2 Measurements

3.2.1 Model comparison

3.2.2 Benchmarks

Chapter 4

Conclusion

Bibliography

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. JTorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification VIII*, 1988.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[6] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[7] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006.

[8] M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[9] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[10] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[11] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is different and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[12] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems*. Springer-Verlag Berlin Heidelberg, 2005.

[13] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *National Bureau of Standards, Special Publication*, 1982.

[14] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[15] Glenford J. Myers and Corey Sandler. *The Art of Software Testing.* John Wiley & Sons, 2004.

[16] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[17] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[18] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap).* Uitgeverij Tutein Nolthenius, 1995.

[19] S.C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64 –73, nov 1997.

[20] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003. University of Southampton.

[21] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[22] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[23] Floor Sietsma. A case study in formal testing and an algorithm for automatic test case generation with symbolic transition systems. Master's thesis, Universiteit van Amsterdam, 2009.

[24] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[25] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[26] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[27] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.